ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/252309999

Forensic Data Recovery from Flash Memory

Article *in* Small Scale Digital Device Forensics Journal · January 2007

CITATIONS
191

READS
12,848

5 authors, including:

Coert Klaver
The Netherlands Forensic Institute
11 PUBLICATIONS **970** CITATIONS
SEE PROFILE

Ronald van der Knijff
Netherlands Forensic Institute
6 PUBLICATIONS **285** CITATIONS
SEE PROFILE

Mark Roeloffs
Netherlands Forensic Institute
7 PUBLICATIONS **250** CITATIONS
SEE PROFILE

All content following this page was uploaded by Coert Klaver on 08 January 2016.

The user has requested enhancement of the downloaded file.

Forensic Data Recovery from Flash Memory

Marcel Breeuwsma, Martien de Jongh, Coert Klaver, Ronald van der Knijff and Mark Roeloffs

***Abstract*—Current forensic tools for examination of embedded systems like mobile phones and PDA's mostly perform data extraction on a logical level and do not consider the type of storage media during data analysis. This paper suggests a low level approach for the forensic examination of flash memories and describes three low-level data acquisition methods for making full memory copies of flash memory devices. Results are presented of a file system study in which USB memory sticks from 45 different make and models were used. For different mobile phones is shown how full memory copies of their flash memories can be made and which steps are needed to translate the extracted data into a format that can be understood by common forensic media analysis tools. Artifacts, caused by flash specific operations like block erasing and wear leveling, are discussed and directions are given for enhanced data recovery and analysis on data originating from flash memory.**

***Index Terms*—embedded systems, flash memory, physical analysis, hex analysis, forensic, mobile phones, USB sticks.**

I. INTRODUCTION

THE evolution in consumer electronics has caused an exponential growth in the amount of mobile digital data. The majority of mobile phones nowadays has a build in camera and is able to record, store, play and forward picture, audio, and video data. Some countries probably have more memory sticks than inhabitants. A lot of this data is related to human behavior and might become subject of a forensic investigation.

Flash memory is currently the most dominant non-volatile solid-state storage technology in consumer electronic products. An increasing number of embedded systems use high level file systems comparable to the file systems used on personal computers. Current forensic tools for examination of embedded systems like mobile phones or PDAs mostly perform logical data acquisition. With logical data acquisition it's often not possible to recover all data from a storage medium. Deleted data for example, but sometimes also other data which is not directly relevant from a user standpoint, can not be acquired and potentially interesting information might be missed. For this reason data acquisition is wanted at the lowest layer where evidence can be expected. For hard disk based storage media it's common to copy all bytes from the original storage device to a destination storage device and then do the analysis on this copy. The same procedure is desired for embedded systems with solid-state storage media.

This paper suggests a low level approach for the forensic examination of flash memory. In chapter II the most important technology basics of flash memories are explained. Chapter III describes three low-level data acquisition methods for flash memories, first with so called *flasher tools*, then by usage of an access port commonly used for testing and debugging and finally with a semi-invasive method where the flash memory chips are physically removed from the printed circuit board. Chapter IV explains methods to translate the extracted data to file system level where common forensic media analysis tools can be used for further analysis. Experimental results are given on data originating from USB sticks and mobile phones. Chapter V explains some artifacts characteristic to data originating from flash file systems.

II. FLASH TECHNOLOGY

Flash memory is a type of non-volatile memory that can be electrically erased and reprogrammed. Flash memory comes in two flavors, NOR[1] flash and NAND[2] flash, named after the basic logical structures of these chips. Contrary to NAND flash, NOR flash can be read byte by byte in constant time which is the reason why it is often used when the primary goal of the flash memory is to hold and execute firmware[3], while parts of NOR flash that are not occupied by firmware can be used for user data storage. Most mobile media, like USB flash disks, or multimedia centred devices like digital camera's and camera phones, use NAND flash memory to create compact mobile data storage. This chapter explains the basics of flash technology first on the physical level and then from a logical perspective. An introduction to NAND flash memory can be found in [5], more in depth information can be found in [9].

A. Physical Characteristics

The physical mechanism to store data in flash memory is based on storing electrical charge into a floating gate of a transistor. This charge can be stored for extended periods of time without using an external power supply but gradually it will leak away caused by physical effects. Data retention specifications for current flash memory are between 10 and 100 years.

Flash memory can be written byte for byte, like EEPROM[4], but it has to be erased in blocks at a time before it can be re-written. Erasing results in a memory block that is filled completely with 1's. In NAND flash, erase blocks are divided further into pages, for example 32 or 64 per erase block. A page is usually a multiple of 512 bytes in size, to emulate 512 byte sector size commonly found in file systems on magnetic media. Additionally, a page has a number of so called 'spare area' bytes, generally used for storing meta data. Some flash disk drivers use the concept of zones[5]. A zone is a group of blocks, usually 256 to 1024. Contrary to blocks and pages, a zone is just a logical concept, there is no physical representation. See figure 1 for a dissection of NAND flash memory.

---

[1] NOR flash memory was introduced in 1988 by Intel.

[2] NAND flash memory was introduced in 1989 by Toshiba.

[3] Firmware is software that is embedded in a hardware device (like a mobile phone or a PDA).

[4] Electrically Erasable Programmable Read Only Memory.

[5] The term partition is sometimes also used to indicate sections of flash memory.

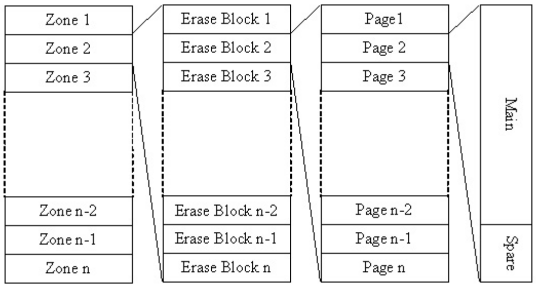


Fig. 1. Dissection of NAND flash memory

TABLE I
EXAMPLE SPARE AREA SIZES FOR DIFFERENT PAGE SIZES (IN BYTES)

Page Size	**Spare area size**	**Total page size**	**Block size**
256	8	264	8448
512	16	528	16896
2048	64	2112	135168

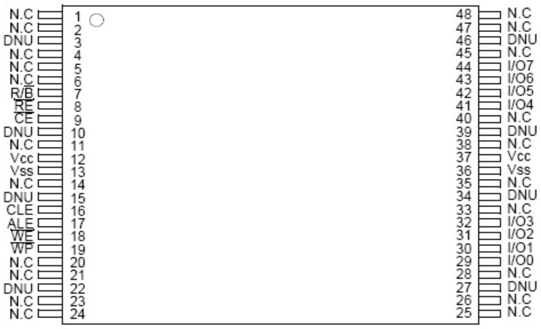


Fig. 2. Typical electrical interface of a NAND flash chip

TABLE II
PIN NAMES OF A NAND FLASH CHIP

Pin	**Description**
CLE	COMMAND LATCH ENABLE
ALE	ADDRESS LATCH ENABLE
CE	CHIP ENABLE
RE	READ ENABLE
WE	WRITE ENABLE
WP	WRITE PROTECT
R/B	READY/BUSY OUTPUT
PRE	POWER-ON READ ENABLE
Vcc	POWER
Vss	GROUND
N.C	NO CONNECTION

Cycle	**I/O$_0$**	**I/O$_1$**	**I/O$_2$**	**I/O$_3$**	**I/O$_4$**	**I/O$_5$**	**I/O$_6$**	**I/O$_7$**	
1	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	Column Address
2	A_8	A_9	A_{10}	A_{11}	L	L	L	L	Column Address
3	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}	A_{18}	A_{19}	Row Address
4	A_{20}	A_{21}	A_{22}	A_{23}	A_{24}	A_{25}	A_{26}	A_{27}	Row Address

TABLE III
TYPICAL ADDRESSING CYCLES FOR A NAND FLASH CHIP

Each page has an area of bytes, often referred to as the redundant area or spare area. Table I shows spare area sizes for different page sizes. The spare area can contain information on the status of the block or the page. For instance when a block turns bad, it will be marked here. The spare area can also contain ECC[6] data. ECC data is used to detect errors in a page. With the ECC data an error of one bit can be corrected, after which the block will be marked bad. Finally the spare area can contain information necessary for the physical to logical address mapping

Erasing a block causes a block to deteriorate. Blocks can be erased between 10^4 and 10^6 times before bits in this block start to become inerasable (stay '0'). Such a block is then called a 'bad block'. NAND flash usually already has bad blocks when leaving the factory. In datasheets of NAND flash chips, the guaranteed minimal number of good blocks when first shipped is specified. Typically at least 98% of the blocks are guaranteed to be in working order. Initial bad blocks are marked as such in the spare area.

In order to spread the erasing of blocks as evenly as possible over the full range of physical blocks, flash memory vendors have developed so called 'wear levelling' algorithms [6] [7]. The idea is that spreading the wear, caused by erasing a block, as much as possible over the whole capacity of the flash memory will increase the overall lifetime of the memory. For manufacturers of memory devices, the wear levelling algorithm can be very sensitive intellectual property, so any inquiries that look like questions about the wear levelling algorithm will often be left unanswered.

However, for the reconstruction of data in a flash memory, it is not necessary to know *how* the wear levelling created the physical image that is copied of a flash chip. All one needs to know is how to recreate the right order of physical blocks in order to create a logical copy of the higher level file system. In other words: wear leveling can be seen as a dynamic process that rearranges pages and/or blocks continuously in order to extend flash lifetime. When trying to interpret a static 'snapshot' of the wear leveling process (the exact binary copy of the physical flash memory at one particular moment) no knowledge of the 'dynamic behavior' of the wear leveling algorithm is needed.

The electrical interface of NAND flash differs from that of RAM. NAND flash has a multiplexed address/data bus, generally referred to as the I/O (Input/Output) lines. This bus can be either 8 or 16 bits wide. An example of the electrical interface of a NAND flash chip is shown in figure 2, with the pin names in table II. Data in the NAND flash chip is accessed by first applying the address of the required data on the I/O lines. As the highest address is generally higher than can be reached with 8 or 16 I/O line bits, the address is latched into the chip in three to five address cycles. After the address is latched into the chip, the data can be clocked out over the same I/O lines. A typical sequence to get access to data in a NAND flash chip is shown in table III.

Further explanation of the inner workings of flash memory and the differences between NAND and NOR flash is beyond the scope of this article.

B. Logical Characteristics

There are several ways in which flash memory can be used as file storage in embedded systems. Three of them are explained below. A simplified diagram of components involved in host Operating System (OS) access to a flash file system is shown in figure 3. As a reference, the situation for a hard disk is shown on the left hand side. In case of a hard disk, the host OS accesses the hard disk through the file system driver (FSD). The FSD issues commands to the hard disk, for instance the ATA[7] command 'Read Sector' to read data from a Logical Block Address (LBA[8]). See [8] for more information on ATA commands.

---

[6] Error Checking and Correcting.

A USB flash disk presents itself to its host as a storage device. After mounting, the host OS can access the device. On the 'Wintel' platform for example, a new drive is created when a USB flash disk is inserted into a USB port, after which files can be accessed. The disk access commands issued by the FSD are channelled through USB to the USB flash disk. The USB flash disk controller interprets these commands and accesses data in flash memory. To manage the special properties of flash memory the controller generally stores additional information[9] with that data. For instance, the LBA in the ATA command will not be the same as the physical address in the flash chip where the data is actually stored. Information necessary for mapping a LBA to a physical location is stored in the flash memory chip as well.

An embedded system, like a mobile phone or a digital camera, can use a similar mechanism, see figure 3, 'embedded device 1'. When the embedded system is connected to a host, the host OS can access the devices flash file system through standard disk access commands. The devices OS (Windows CE, Symbian, proprietary) receives the disk access command from the host OS file system driver and returns the requested data. In this way, the host OS doesn't see any difference between a hard disk or an attached device.

Another way of accessing flash based storage in an embedded device is shown in figure 3, 'embedded device 2'. Here the flash file system is accessed through a proprietary application that runs on the host OS. The application communicates with the embedded system and presents the data in the flash file system on the host. An example of this mechanism is access to a disk on a windows CE device through ActiveSync. ActiveSync makes use of the Remote Application Programming Interface (RAPI)[10] to get data, such as files on the devices file system, to the host. Although in this case the file system on the device can be viewed in the same way as other file systems on the host (with windows explorer), the higher level flash file system on the device might be of a completely different structure that usual in magnetic media.

III. DATA ACQUISITION

The first principle when examining electronic evidence is to keep data held on a storage medium unchanged. For embedded systems this principle is more challenging than it looks at first sight. Issues like network connections are similar to the open systems world although it might be more difficult to detect that an embedded system is connected to other systems. For flash memory wear leveling might cause unpredictable data changes. Switching mobile phones off and/or on has shown data changes probably caused by wear leveling and/or garbage collection algorithms. More research is needed on this topic but for now the general rule is to keep the number of power cycles as low as possible.

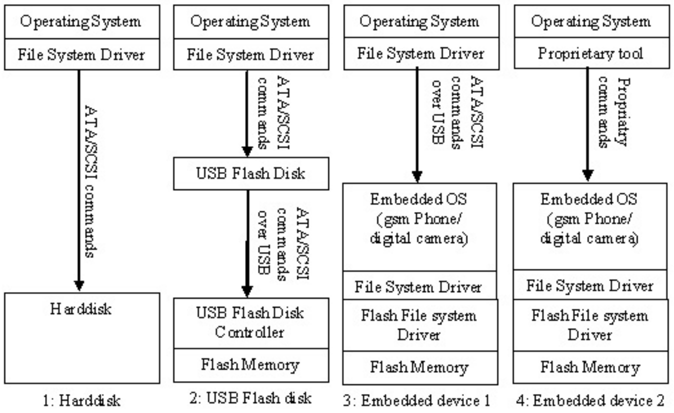


Fig. 3. Components involved in hard disk and flash memory access

In this chapter three possible data acquisition approaches are presented for obtaining a full copy of flash memory data. flasher tools are discussed first, followed by a method using the JTAG[11] test access port of an embedded device. Finally the most invasive method is described in which the flash chip is physically removed and read with an external reader.

A. *Flasher Tools*

The most easy and non-invasive way to read flash data is by using a simple hardware interface and software that copies all flash memory data from the target system to another system for further analysis. Unfortunately there's no general method for this procedure because every embedded system can have its own dedicated interface to data stored in flash memory chips. There's also no standardized "embedded system operating system" with documented low level flash memory access functions. However, memory copying tools specifically targeted towards a certain device (range) exist and can sometimes be used for forensic purposes. These tools mainly originate from two sources: manufacturers or service centers who use these tools for debugging and diagnostics and sometimes for in field software updates, and hackers who use these tools for checking and changing device functionality[12].

Great care should be taken when using these tools for forensic examinations. Besides memory copying functionality these tools sometimes have other options[13] which are devastating

---

[7] AT Attachment (ATA) is a standard interface for connecting storage devices such as hard disks and CD-ROM drives inside personal computers.

[8] Logical Block Address, the address of data by the linear mapping of storage units.

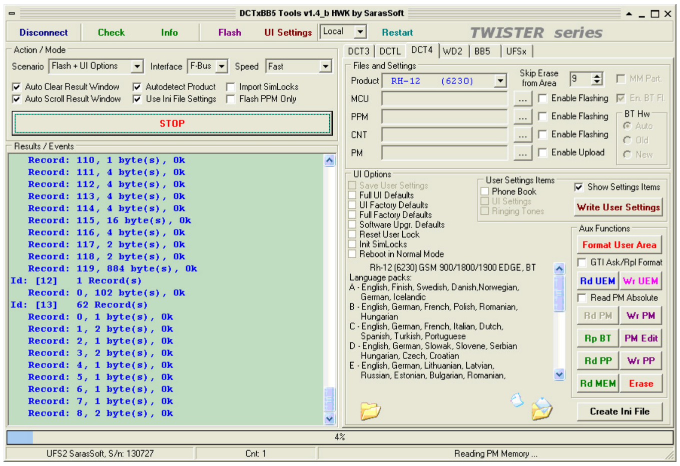
[9] Also called meta data.

[10] msdn.microsoft.com/library/en-us/wceactsy/html/cerefRAPIReference.asp.

[11] JTAG: Joint Test Action Group, see "IEEE Std 1149.1 Standard Test Access Port and Boundary-Scan Architecture".

[12] In the mobile phone area these hacker tools are used for example to change the IMEI, to remove lock codes or for upgrading or debranding.

[13] Like writing or erasing memory, changing serial numbers or adding functionality.



Local

Scenario: Flash + UI Options
Interface: F-Bus
Speed: Fast

☑ Auto Clear Result Window
☑ Auto Scroll Result Window
☑ Autodetect Product
☑ Use Ini File Settings
☐ Import SimLocks
☐ Flash PPM Only

```
   Record: 110, 1 byte(s), Ok
   Record: 111, 4 byte(s), Ok
   Record: 112, 4 byte(s), Ok
   Record: 113, 4 byte(s), Ok
   Record: 114, 4 byte(s), Ok
   Record: 115, 16 byte(s), Ok
   Record: 116, 4 byte(s), Ok
   Record: 117, 2 byte(s), Ok
   Record: 118, 2 byte(s), Ok
   Record: 119, 884 byte(s), Ok
Id: [12]   1 Record(s)
   Record: 0, 102 byte(s), Ok
Id: [13]   62 Record(s)
   Record: 0, 1 byte(s), Ok
   Record: 1, 2 byte(s), Ok
   Record: 2, 1 byte(s), Ok
   Record: 3, 2 byte(s), Ok
   Record: 4, 1 byte(s), Ok
   Record: 5, 1 byte(s), Ok
   Record: 6, 1 byte(s), Ok
   Record: 7, 1 byte(s), Ok
   Record: 8, 2 byte(s), Ok
```

Product: RH-12 (6230)
MCU:
PPM:
CNT:
PM:
Skip Erase from Area: 9
☐ Enable Flashing
☐ Enable Flashing
☐ Enable Flashing
☐ Enable Upload
☐ MM Part.
☑ En. BT Fl.
BT Hw: ● Auto ○ Old ○ New

☐ Save User Settings
☐ Full UI Defaults
☐ UI Factory Defaults
☐ Full Factory Defaults
☐ Software Upgr. Defaults
☐ Reset User Lock
☐ Init SimLocks
☐ Reboot in Normal Mode

☐ Phone Book
☐ UI Settings
☐ Ringing Tones
☑ Show Settings Items
☐ GTI Ask/Rpl Format
☐ Read PM Absolute

Rh-12 (6230) GSM 900/1800/1900 EDGE, BT
Language packs:
A - English, Finish, Swedish, Danish,Norwegian, German, Icelandic
B - English, German, French, Polish, Romanian, Hungarian
C - English, German, French, Italian, Dutch, Spanish, Turkish, Portuguese
D - English, German, Slowak, Slovene, Serbian Hungarian, Czech, Croatian
E - English, German, Lithuanian, Latvian, Russian, Estonian, Bulgarian, Romanian,

4%

UFS2 SarasSoft, S/n: 130727 | Cnt: 1 | Reading PM Memory ...

Fig. 4. Screenshot of twister series flasher box software

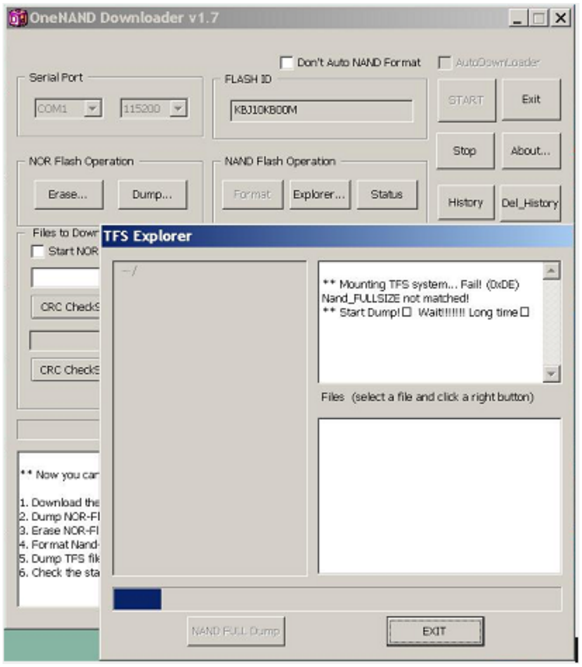
in a forensic context. Before usage on an exhibit a flasher tool needs to be tested on a similar device from a reference collection: once thoroughly to check the functionality, and preferably before each individual examination to train the examiner in using only the forensically sound options of such a tool.

For mobile phones a lot of these tools can be found on the internet in forums like *gsm-forum* [23] or online shops like *gsmtechnology* [24] or *gsm-server* [25]. Flasher boxes are mostly accompanied by a large number of cables to connect different phone models.

An example of a flasher box with software which can be used with a wide variety of phones is the *Twister flasher box*. Figure 4 shows a screenshot of the *Twister series* flasher box software. With this software it is possible to make full memory copies of a large range of Nokia models. For some models only partial memory copies are possible. Figure 5 shows a screenshot of a tool for making complete flash memory copies of Samsung D500/D600 handsets. Recent versions of this tool also copy the meta data needed for reconstruction of the file system (see section IV-B). A lot of these flasher tools work in a similar way: they enter the bootstrap mode of a phone; upload dedicated flash loader software to RAM; execute this software and then use it for low level access to the flash memory. Further research is needed to incorporate this mechanism in future forensic mobile phone acquisition software.

1) advantages and disadvantages:

- Hardware connection is usually easy with a connector.
- Flash memory can be imaged without de-soldering of flash memory chips.
- Some tools do not make a full forensic image of flash memory (some do only parts of the memory space or skip spare area).
- It can not be guaranteed that no data is written in flash memory.



☐ Don't Auto NAND Format
☐ AutoDownLoader
COM1
115200
KBJ10KB00M
☐ Start NOR

** Now you car

1. Download the
2. Dump NOR-Fl
3. Erase NOR-Fl
4. Format Nand-
5. Dump TFS fil
6. Check the sta

~/

```
** Mounting TFS system... Fail! (0xDE)
Nand_FULLSIZE not matched!
** Start Dump!□ Wait!!!!!! Long time□
```

Files (select a file and click a right button)

Fig. 5. D500 OneNAND Downloader

B. JTAG

When a forensically sound image cannot be produced with flasher tools, a second option is to use a JTAG[14] test access port of an embedded device. A JTAG test access port is normally used to test or debug embedded systems but can also be used to access flash memory [11].

This section explains about two test modes (extest[15] and debug mode) and how to use these test modes for forensic imaging of flash memory. JTAG enabled boards have extra test pads, usually not directly reachable for the user. The second part of this section describes a method to find this JTAG test access port on an embedded system with unknown layout.

1) How to access flash memory using JTAG: Flash memory chips are not JTAG enabled. But, as shown in an example embedded system in figure 6, flash memory chips are usually connected to other chips like a processor. This processor can be used to gain access to flash memory if the processor is JTAG enabled. Most JTAG enabled processors offer an extest mode or debug mode. Note that extest or debug mode may not be available on all processors and some processors offer both modes. The next two paragraphs explain how to use these two modes for forensic imaging of flash memory.

a) Extest mode: In extest mode, all processor pins are controlled by a JTAG controller while the processor core is disabled. Test vectors are loaded or read using a, usually, long shift register. An external flash memory can be read by loading and reading a series of test vectors. An example in figure 7 shows how to access a NOR flash memory using extest mode and a series of two test vectors.

---

[14] JTAG: Joint Test Action Group, see "IEEE Std 1149.1 Standard Test Access Port and Boundary-Scan Architecture".

[15] Extest: External test mode.

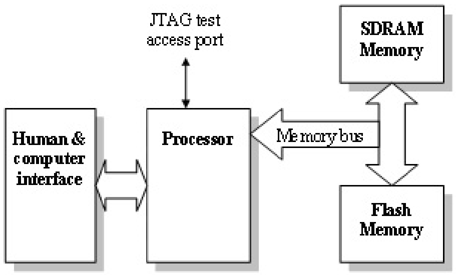


Fig. 6. An example of an embedded system

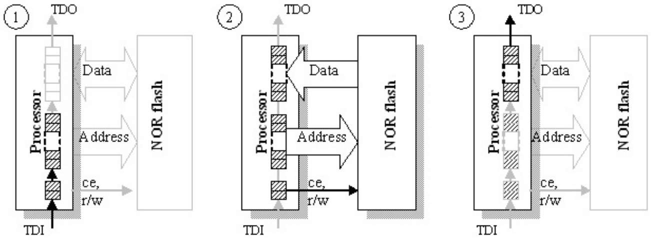


Fig. 7. Using extest mode for accessing memory

1) The first test vector contains an address of a NOR flash memory location and also control-signals (ce, r/w) with a read command. This test vector is activated after loading. See step 1 in figure 7.
2) After an access time, the flash memory chip responds with the requested data on the data bus and is captured in a second test vector. See step 2 in figure 7.
3) This second test vector is read by a PC and the data from the data bus is stored in a file. See step 3 in figure 7. An image of a NOR flash memory chip can be produced by repeating these three steps for all memory locations.

Also NAND flash memory chips can be imaged using extest mode. However this will be slower because it takes a higher number of test vectors to read a byte or word of data from a NAND flash memory. Especially an address latch cycle takes more test vectors compared to a NOR flash memory.

b) Debug mode: Debug circuitry build in a processor can be used to debug embedded software running on this processor. JTAG circuitry in the processor has extra registers to stop and start execution, read status registers or to write and read data from external memory chips. This last option can be used for producing an image of NOR flash memory. A commercially available debugger like *JTAGjet* from *Sygnum systems* [15] can be used for this task. Producing an image of NAND flash memory cannot be done or is difficult using a commercially available debugger.

2) How to find a JTAG test access port: Before an image of flash memory can be produced the JTAG test access port has to be found in an embedded system. On some PCB's[16] the test pads of the JTAG test access port are located in a row and clearly marked, but usually they look similar to other test pads and may even be spread over both sides of the PCB, making it difficult to find them between all other test pads.

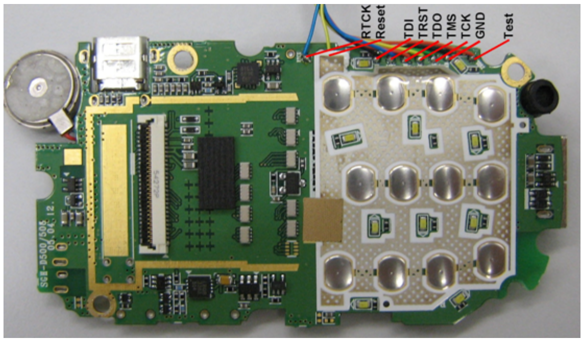


Fig. 8. JTAG test access port of a Samsung SGH-D500

When a manufacturer of an embedded system cannot or does not want to give information about its JTAG test access port, a forensic examiner could try to find the JTAG test access port. This section explains some methods to find a JTAG test access port.

1) Modern embedded systems use a processor chip build in a micro BGA[17] casing. A way to find the JTAG test access port is to de-solder the processor chip of a reference embedded system. Traces on the PCB can be measured with a multi-meter to find the JTAG test access port. This method needs the availability of a reference embedded system with an equal PCB layout and usually leads to the destruction of this reference embedded system.
2) Most embedded systems use a multi-layer PCB. A multi-layer PCB can be viewed with an X-ray machine. The traces can be followed by focusing on the right layer. However, parallel running tracks in different layers and components on both sides of the PCB mostly thwart the attempts to follow the interesting connections.
3) Measure all test pads on the PCB. Because JTAG inputs and output have special properties it is possible to find the JTAG test access port between all other test pads. First a simple measurement has to be done on all test pads of a reference embedded system. This first step eliminates most test pads and can be done relatively fast, although the number of test pads can be high (>100) on some embedded systems. This measurement leads to a limited number of candidate test pads (test pads belonging to the JTAG test access port). The test access port can be found with a second measurement by testing all possible input / output combinations with an exhaustive search algorithm until a valid signal is received.

Figure 8 shows an example of a JTAG test access port on a SGH-D500 from manufacturer Samsung. This is an example where the test pads of the JTAG test access port are located in a row.

3) JTAG advantages and disadvantages:

- The risk of changing data is minimized. It can be guaranteed that no data is written in extest or debug mode. However there is always at least a short period between

---

[16] PCB: Printed Circuit Board.

[17] BGA: Ball Grid Array.

power-up and the time when debug or extest mode is entered. In this period the system itself can interact with flash memory
- Flash memory can be imaged without de-soldering of flash memory chips.
- A complete forensic image can be produced (all data, inclusive spare area, bad blocks etc).
- A disadvantage is that communication in extest mode is slow. Debug mode can be faster however.
- A JTAG test access point can be difficult to find.
- Not all embedded systems are JTAG enabled.

C. Physical Extraction

Another way to produce an image of flash memory is to physically remove a flash memory chip from a PCB and read this flash memory chip with a memory chip programmer or reader. This method can be used when JTAG is not available and software tools can not be used.

This section starts with a description of methods to de-solder chips from a PCB. A chip usually has to be prepared for further processing (cleaning and restoring connections) after removing. This is discussed in the second part of this section. The third part of this section describes how to read flash memory chips with a programmer or reader and discusses the sockets to use.

1) De-soldering of flash memory chips: Most chips are packed in a TSOP[18] or micro BGA[19] casing nowadays. This paragraph discusses methods to remove these chips because special de-soldering equipment is needed for de-soldering to prevent damaging of chips and therefore loss of data. Especially chips packed in micro BGA casings need special care.

a) How to remove TSOP chips: A *NAND512* from manufacturer *ST(SGS-Thomson)* is an example of a flash memory chip packed in a TSOP casing. Although not preferred, TSOP chips can be removed from a PCB with a soldering iron. This method is not preferred because a lot of solder has to be applied on the pins of the chip. It takes a long time to clean the chip afterwards. A better way of removing a TSOP chip is with hot air. Figure 9 shows how this method works. Hot air is blown on the edges of a TSOP chip. Therefore the temperature of the chip itself stays lower than the temperature of the solder connections. When the solder is melted a vacuum air gripper pulls the chip off.

b) How to remove micro BGA chips: An example of a flash memory chip in micro BGA casing is RD28F6408W18 from manufacturer Intel. This example has 56 balls.

Micro BGA chips can be removed with hot air using a rework station. A rework station uses a temperature profile to remove a chip. The temperature has to be hot enough to melt the solder. But be careful: The chip may be damaged if the temperature is too high. The reader is referred to [13] for more information about rework temperature profiles. Especially lead free chips must be handled carefully because lead free solder has a higher melting temperature. The temperature profile is different for each chip and PCB because the convection of heat is subject to many parameters like the thickness of the PCB, the number of layers, the size of the nozzle and the chip size. Always practice on a reference model before removing a chip from an exhibit and use a temperature sensor mounted at the side of the flash memory chip for temperature logging. An example of a rework station is a *TF2000* from manufacturer *Pace* [14]. This rework station can also be used to replace chips.

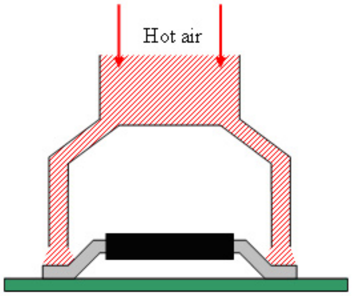


Fig. 9. Removing TSOP chips with hot air

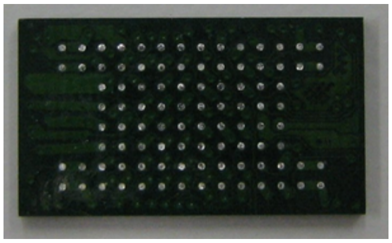


Fig. 10. Example of a micro BGA chip (6408W18B from Intel)

2) How to prepare a chip for further processing: Pins of a TSOP chip can be cleaned with solder wick and flux remover. Make sure that the pins are nicely aligned and no old solder is left behind on the pins. If a micro BGA chip is removed from a PCB the balls on the chip are damaged. Some solder is left behind on the chip and the rest is left behind on the PCB. The result is that the balls have different sizes. These differences in size are a problem for most sockets. These sockets are designed for virgin chips with balls of equal size resulting in bad connections when an unprepared micro BGA chip is used.

One solution for this problem is to repair the balls of the chip. This process is called reballing. Although other methods are available the best method is to use a reballing machine. This machine puts little balls of solder on the connection and locally melts it with a laser beam, but this machine is very expensive. A solution to omit reballing is to use a socket with little springs also called *pogo pins*. The chip is pressed onto the springs and the springs correct the difference in height of the balls. A solution with a socket with pogo pins is preferable because this saves a reballing step and the risk of losing data is minimized.

After cleaning, the flash memory chip can be read with a programmer or reader. This memory chip programmer or

---

[18] TSOP: Thin Small-Outline Package.

[19] BGA: Ball Grid Array.

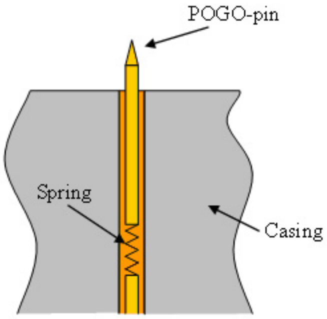


Fig. 11. Pogo-pin

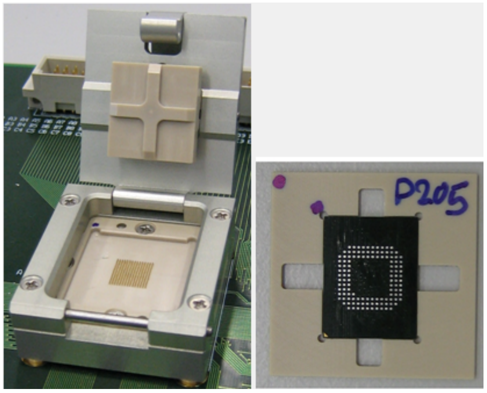


Fig. 12. Universal socket (left) and locator with chip (right)

reader usually has several types of ZIF[20] sockets for connecting a memory chip to the programmer or reader. Flash chips in TSOP casing usually use a casing with 48 pins. Therefore most TSOP chips can be read with only one type of socket. Micro BGA chips, however, are found in many different sizes and differ greatly in number of balls. Usually chips have casings between 40 balls up to 167 or more. The number of sockets to be used is huge (>40) and continues to grow. Usually a socket is expensive and has a long delivery time. Therefore it is not feasible to buy a new socket for each type of chip.

A solution for this problem is to use a socket that can be adapted for many types of chip casings [18]. The solution presented in this paper uses a matrix of 15 x 15 pogo pins where sockets are available for 0.5mm, 0.75mm and 0.8mm pitch. The flash memory chip is held into position by a locator. This locator is specific for each type of casing and can be made relatively fast and easily with a milling machine and is cheap. See figure 12.

3) Flash memory chip programmer or reader: A flash memory chip can be read with a commercially available memory chip programmer like *BP 1600* from manufacturer *BP Microsystems* [16]. A disadvantage is that a driver is needed for each type of memory chip. If a driver for a certain type of chip is not available, the manufacturer of the programmer has to program this driver. This can take some time and is not always possible when a datasheet is not available for example.

Another solution is to use a universal flash chip reader. This custom made design is called 'NFI memory toolkit'. A schematic is drawn in figure 13.

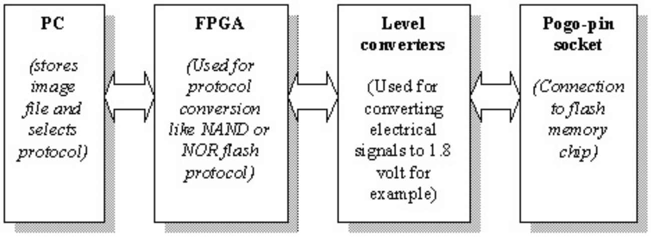

PC
(stores image file and selects protocol)
FPGA
(Used for protocol conversion like NAND or NOR flash protocol)
Level converters
(Used for converting electrical signals to 1.8 volt for example)
Pogo-pin socket
(Connection to flash memory chip)

Fig. 13. Schematic of NFI memory toolkit

An FPGA is used for communicating with a flash memory chip where configurations are available for a NAND and NOR flash protocol (with multiplexed and de-multiplexed address bus). All parameters, like address bus size and data bus size are fully customizable by the PC software. In case of a NOR flash memory a data structure is read from the NOR flash memory (CFI[21] data structure). This data structure contains all parameters needed for reading that particular flash memory (like protocol, memory size etc). The command to read this data structure is compatible with all protocols and the toolkit software automatically uses the parameters to read a NOR flash chip without any configuration from the user. NAND flash chips can also be read automatically because the number of protocols used by NAND flash chip is very limited. The toolkit software automatically scans all protocols until a correct response is received from the NAND flash chip. Due to the automatic configuration properties of the software it is sometimes possible to read flash chips even if a datasheet is not available.

4) Advantages / disadvantages of physical extraction:

- It can be guaranteed that no data is written in flash memory because the embedded system stays powered down.
- Data from broken or damaged embedded systems can be recovered.
- A complete forensic image can be produced (all data, inclusive spare area, bad blocks etc).
- A disadvantage is that there is a risk of damaging the flash memory chip due to the heat for de-soldering.
- The embedded system has to be opened to reach and desolder flash memory chips.

IV. FILE SYSTEM ANALYSIS

Data acquisition as described in the previous chapter results in one or more binary files containing linear bitwise copies of flash memory data. Before any volume analysis and succeeding file system analysis can take place, the sectors of data as used by the high level file system need to be placed in the right order. For devices with a flash file system (FFS[22]) this means finding out how the FFS maps physical data to logical data and how the difference between valid and invalid data can be determined. The result of flash file system analysis is a method that splits the physical data into two parts: a file with all logical sectors in the right order belonging to the actual high level file system and a file with all other data not belonging to the (current) high level file system.

---

[20] ZIF: Zero Insertion Force.

[21] CFI: Common Flash Interface.

[22] Current definitions of a flash file system are a bit greedy. In this paper the term flash file system (FFS) is used to describe data translation mechanisms between the physical flash chip and the file system API of the host operating system. This covers flash translation layers (FTL) of disk-drive emulators and dedicated flash optimized file systems.

The file system analysis process is explained in the next sections in the context of USB memory sticks and for mobile phones.

A. File System Analysis on USB Memory Sticks

The flash file systems on USB memory sticks are usually relatively simple mechanisms that only translate the Logical Block Number used in high level file systems to a low level physical address and do not support wear levelling. In the file systems described in this chapter, the block size of the flash file system is equal to the erase block size of the flash memory chip used. This means that when a logical block changes, the new version is stored in a new erase block and the complete old version is erased. This is not a very efficient way of dealing with flash memory, because pages within an erase block that are not changed are still copied to the new block and the old page is erased, yielding a higher wear that absolutely necessary for this block.

In the USB memory sticks studied for this chapter, the concept of zones is used in two devices[23]. In these devices, a zone is defined as a group of erase blocks, for example: 1024 erase blocks are grouped into a zone. Within this zone, 1000 blocks are actively used to store the high level file system, 24 blocks are kept aside to replace bad blocks when they arise. These blocks are marked in a special way, so that they can be recognised by the controller as such.

For the study on which this section is based a reference collection of USB memory sticks was needed. To create this collection, colleagues at the NFI were asked whether they owned any USB memory and whether they wanted to trade it for a new 128 Mbyte device. This resulted in 45 sticks of different make and model.

1) Identification of controller and memory chips: The controller chips found in USB sticks are sometimes hard to identify, mainly because often only a manufacturer logo and a part number can be found on the chip. But even with this information and some creative Internet queries the manufacturer of the controller can be identified. The memory chips are often much easier to identify. The memory chip usually carries the manufacturer name and logo and the part number. Furthermore, contrary to controller chips, flash chips appear to be produced by companies well known in the electronics industry.

In the reference collection of 45 USB memory sticks, 16 different manufacturers of controllers have been identified, who produced 24 different controllers. Table IV shows all controllers identified in the reference collection. In the reference collection, 8 different manufacturers of NAND flash memory have been identified, who produced 26 different types of chips. Table V shows all unique memory chips identified in the reference collection.

TABLE IV
CONTROLLER CHIPS IDENTIFIED IN THE REFERENCE COLLECTION

Brand	**Type**	**Company Website**
SSS	6633 6666	www.3system.com.tw
ALCOR	AU9382 AU9384 AU9385	www.alcormicro.com
ChipsBank	CBM1183	www.chipsbank.com
PQI	CLCPC02	www.pqi.com.tw
M-Systems	T4 Titan	www.m-systems.com
KTC	FC1325N	www.ktc.com.tw
Lexar	FC1610	www.lexar.com
GenesysLogic	GL814	www.genesyslogic.com
OTi	OTi002168 OTi006808 OTi006828	www.oti.com.tw
PointChips	PP2201 PP2366	www.pointchips.com
Silicon Motion	SM3210	www.siliconmotion.com.tw
SONIX	SN11085A SN11088B	www.sonix.com.tw
Trumpion	t33521FL	www.trumpion.com.tw
Prolific	?	www.prolific.com.tw
SanDisk	?	www.sandisk.com
Silicon Integrated Systems Corp.	?	www.sis.com

TABLE V
NAND FLASH CHIPS, IDENTIFIED IN THE REFERENCE COLLECTION

Brand	**Type**	**Company Website**
Hitachi	HN29V51211T-50	www.hitachi.com
Hynix	HY27UF081G2M-TPCB HY27US08121M-TCB HYF33DS512800ATCG1	www.hynix.com
Samsung	K91FG08U0M-YBB0 K9F1208UOM-YCB0 K9F1G08U0A-PCB0 K9F1G08U0M-VIB0 K9F1G08U0M-YCB0 K9F1G16U0M-YCB0 K9F2808U0B-YCB0 K9F2808U0C-YCB0 K9F2808U0M-YCB0 K9F5608U0A-YCB0 K9F5608U0B-YCB0 K9F5608U0C-YCB0 K9K1G08U0M-YIB0	www.samsung.com
RF	N1208U0B-0FF	?
ST	NAND256W3A0AN6 NAND512W3A0AN6	www.st.com
SanDisk	S4164901	www.sandisk.com
Toshiba	TC58128FT TC58DVG02A1FT00 TC58DVG04B1FT00 TC58DVM92A1FT00	www.toshiba.com

To put the number of different memory chips in perspective, most NAND flash memory chips are compatible but some important properties are differing. Some of these properties are:

- Storage capacity: 16 Mbyte to 128 Mbyte
- Number of addressing cycles: three, four or five
- Width of the I/O bus: 8 or 16 bit
- Operating voltages: 1.70~1.95V, 2.4~2.9V, 2.7~3.6V
- Erase block size: 16 kbyte, 128 kbyte

---

[23] Smart Media format and the Alcor 9385 controller format.

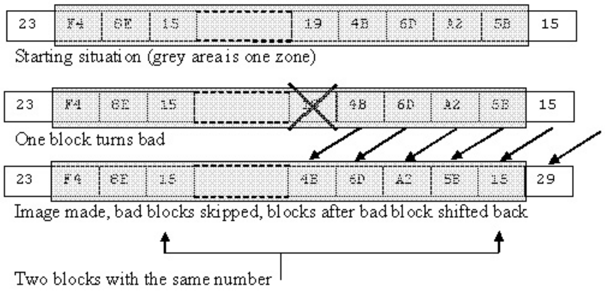

23
F4
8E
15
19
4B
6D
A2
5B
15
Starting situation (grey area is one zone)
One block turns bad
29
Image made, bad blocks skipped, blocks after bad block shifted back
Two blocks with the same number

Fig. 14. Block shifting into previous zone because of bad block skipping

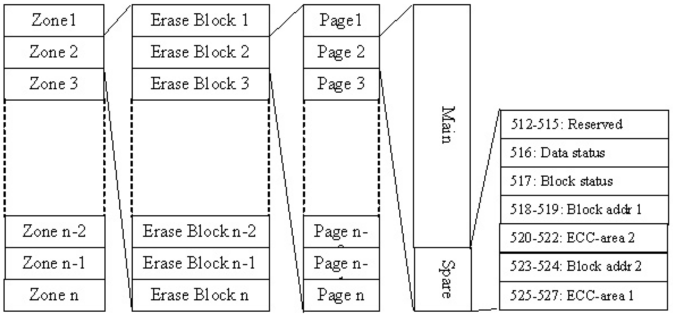


Fig. 15. Smart Media format

TABLE VI
BLOCK ADDRESS INSIDE THE BLOCK ADDRESS FIELDS

Byte	D7	D6	D5	D4	D3	D2	D1	D0
518, 523	0	0	0	1	0	BA9	BA8	BA7
519, 524	BA6	BA5	BA4	BA3	BA2	BA1	BA0	P

- Page size: 528 byte, 2112 byte
- Housing: TSOP 48

2) Making an exact copy of the flash chip(s): When the chip is extracted from the PCB, it can be read with a device programmer[24] as described in section III-C. When reading the content of flash chips one needs to be aware of the fact that some programmers have a special way of handling NAND flash. When programming a NAND flash in a production environment, the programmer obviously wants to skip bad blocks. Further more, when a file is loaded in the programmer, one wants to be sure that the file will fit in the flash chip, so the programmer will only accept files smaller than the guaranteed minimal number of good blocks. These two properties often also play a role when reading the device. Bad blocks are not read, and only the guaranteed minimal number of good blocks is read. For making a forensically sound copy of a memory chip this is not the desired behaviour.

Skipping of bad blocks can lead to the following problem when reconstructing the high level file system: Suppose a USB memory controller divides the memory into zones of 256 blocks. Each block (belonging to the high level file system) within a zone has to have a unique number, stored in the spare area of each page in that block. Then one bad block in a zone arises. After this, the memory is imaged with a programmer that skips bad blocks. The resulting image is also split up into zones. Now, the zone with the bad block will contain one block of the next zone because all blocks after the bad block will be shifted one block in the image. Now it will be very likely that we have two blocks with the same 'unique' number in one zone. See Figure 14.

Reading up to only the guaranteed minimum number of good blocks can result in blocks at the high end of the memory chip not being read. These blocks might very well contain parts of the high level file system so not reading them might hinder reconstruction of the high level file system.

There are several solutions to these problems. One is to request the manufacturer of the device programmer to make a special version of the algorithm for the specific chip which reads all blocks, good and bad. Another is to develop an 'in house' solution. For the Memory Toolkit, described in paragraph III-C3, an algorithm for reading NAND flash was developed. Furthermore, an adapter socket was made to make contact to TSOP 48 housings. With this system a complete binary copy of a NAND flash memory chip can be made. The rest of this paragraph is based on complete binary copies of flash chips, made with the NFI memory toolkit.

3) Converting the copy to the high level file system: In order to convert the exact copy of the NAND flash memory back to the file system as seen by the host OS, the meta data in the NAND flash memory needs to be interpreted. There are three main questions in this regard:

1) What is the granularity of the flash file system?
2) Where is the meta data stored?
3) How can the meta data be interpreted?

The answers to these questions are of course known by the manufacturer of the USB memory controller. Sometimes the answers can be found in literature, see for instance the definition of the Smart Media File System. When unlucky, the controller manufacturer is unable or unwilling to give information of the flash file system. This leaves reverse engineering of the flash file system as the last option. In this case a reference stick of same make and model is nearly indispensable.

a) Smart media flash file system: The Smart Media format, introduced by Samsung and Toshiba in the late 90's is an example of how to store a FAT file system in flash memory. Information on the Smart media flash file system cannot be found on Samsung's and Toshiba's websites anymore, as the format is 'End Of Life'. Now only copies of the document can be found on Internet [9].

Each FAT cluster is stored in a flash erase block, while the information on which FAT cluster is stored in which flash erase block is kept in the spare area of each page in the erase block. Furthermore, the spare area contains information on the status of the data, the block and error correction code.

BA0 - BA9 are the bits for a nine bit block address, yielding a maximum of 2^{10} * 0x4000 = 16777216 bytes (16 MByte) of data. Smart Media memories with more than 16 MByte, use zone based block management where each zone has 1024 physical blocks in which 1000 blocks are used as logical blocks. **P** is the parity bit for even parity.

---

[24] We initially used a BP1600 from BP Microsystems.

When all bad blocks are skipped all pages within an erase block have the same logical block number (LBN). So to convert a raw Smart Media flash memory copy to the high level file system, one needs to sort all erase blocks to their logical block number (within each zone) and for each page strip off the spare area.

b) Unknown flash file system: In case there is no knowledge about the used flash file system, the flash file system needs to be reverse engineered. A reference device of exact same make and model as the exhibit is making the reverse engineering job a lot easier, but it is indispensable when it comes to validating the method.

4) What is the granularity of the flash file system?: To illustrate a way to find out about granularity of the flash file system, we investigated a USB memory stick with a Lexar FC1610 controller and a K9K1G08U0M-YCB0 128 Mbyte memory by Samsung. The flash chip has a page size of 512+16 bytes, and 32 pages per erase block. To answer the question on granularity, first the USB memory stick is completely filled with files of random length and content. Then a logical image needs to be made of the USB memory stick[25]. When this image is made, two lists can be produced from this image. One containing a hash value[26] over each 512 bytes of the image. Then one containing a hash value over each 16384 bytes. Next, a physical image needs to be made of the flash memory chip (see chapter III). From this image, two more lists need to be made. One with hash values of each page (without the spare area), and one with a hash value of each erase block (without the spare area's).

Load these lists into a tool with which the lists can be sorted on the various columns. In this case Microsoft's Access and Excel[27] were used. When sorting on hash values, it will become clear whether there are identical hash values each page bytes or each block.

In table VII, the first 7 entries in the table are shown. From this it is already clear that on page level identical MD5 hashes are found, so the granularity for this flash file system is page size. Logical sector 0x26193 is stored in (mapped to) physical page 0x141B6, 0x1C10A is stored in 0xB6AD, and so on.

Note that logical sectors 0x01DA and 0x001E have the same MD5 hash, so most probable they have the same content. This can have several causes:

- the test data is not random enough;
- there are identical (bad) blocks that are not changed by the system anymore;
- there are spare blocks (within a zone) that are not yet used to store the high level file system.

Now this mapping is based on sorting the MD5 hashes of content of both physical and logical images. No need to say that when we want to reconstruct the file system from a physical copy, the mapping from physical to logical has to be found in another way. We need to explore the meta data.

TABLE VII
HASH VALUES CALCULATED OVER PAGE AND BLOCK DATA

Logical block (16384 bytes)	**Md5 hash**
00001350	0005D859EC1AA3DDD590A13761CB520A
00000DCB	00186EC8203D6759E9049511E8F5E238
0000008E	001F200A0668AB9071D9E207C9783895
00001C15	0020C85071AA64FF1AF56542CD2D8AA1
00000490	0028EBEDAB6090610EDCFD0F45985F5D
00000E84	002E99C24BC440505E8477811AAB1639
000003B5	0043A7ADF684459073A1E2EE4DAC9161
...	

Physical block (16384 bytes)	**Md5 hash**
00001A43	00081EDB46257F7C61E14A175F638ADF
00001BBA	0018BC3D54C18EDCC54D8CBE344FCAEF
000008CA	0018E4019976D0B0D9984B4816241F9E
0000070B	001FC96E9FB689249EA97B03EB298052
00000BAB	002711850DA93D0707FEF134945C83D3
00001EA2	0027BA1FA4C4F2C60A4926C61E62069F
0000007A	00384D4C7AFE6548C2ACB6A6FD4AB664
...	

Logical sector (512 bytes)	**Md5 hash**
00026193	0000EE07779E4A23827BF396E501B121
0001C10A	000100B88720F47EB8C517AB796D3C20
0000F7A0	0001A198418524B9A30E99CEB0882AB2
000001DA	0001B7FE9BDAD0920FB2A505FBA0B20B
000000E5	0001B7FE9BDAD0920FB2A505FBA0B20B
000374A6	0001DE7FABA54AAEECB6E8E3295D6D32
00007D51	0001F3620D7C07A58050A223236EC1C4
...	

Physical page (512 bytes)	**Md5 hash**
000141B6	0000EE07779E4A23827BF396E501B121
0000B6AD	000100B88720F47EB8C517AB796D3C20
00006E63	0001A198418524B9A30E99CEB0882AB2
00038E5D	0001B7FE9BDAD0920FB2A505FBA0B20B
00038CC8	0001B7FE9BDAD0920FB2A505FBA0B20B
00031AA9	0001DE7FABA54AAEECB6E8E3295D6D32
00026BF4	0001F3620D7C07A58050A223236EC1C4
...	

5) Where is the meta data stored?: Meta data can be stored in the spare areas of the flash memory. In case there is a page size granularity, all spare areas within each block will contain different information. In case there is a block size granularity, spare areas within one erase block may contain at least a few identical bytes: the ones that indicate the logical block number. In section IV-A6 a method is described to analyze meta data stored in spare area's.

Meta data can also be stored in the normal pages/blocks. No generic method has yet been developed for USB memory to analyze this type of meta data storage.

6) How can the meta data in spare area's be interpreted?: To illustrate a way to get information on the meaning of the meta data, we investigated a USB memory stick with an Alcor 9385 controller and a 64 Mbyte NAND512W3A flash chip by ST.

First the granularity of the flash file system of this controller was investigated as described in section IV-A4. It appeared that the granularity is erase block size. This means that the order of pages within an erase block are not changed when stored in physical flash memory.

To get an idea of what data is stored in the spare areas, a table of 16 column by 256 row elements is build. Each column represents one byte location in the spare area's, each row is a possible value of that byte. All spare areas are read and for each byte location in the spare area, the counter in the corresponding value row is incremented. When done, each element contains the frequency of a certain value in a certain

---

[25] For instance with the 'dd' command under linux.

[26] For instance the MD5 hash.

[27] When using Excel, the maximum number of rows is 65536. This is just enough for comparing all sectors of a memory of 32Mbyte or all 4k blocks of a 1 Gbyte memory.

	Spare area byte location										
Byte Value	**0-5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	96	96	608	3538	3427	96	96	608	3455	3508	96
1	0	0	512	0	0	0	0	512	0	0	0
2	0	0	512	0	0	0	0	512	0	0	0
3	0	0	512	3268	3247	6810	0	512	3223	3292	6501
4	0	0	512	0	0	0	0	512	0	0	0
5	0	0	512	0	0	0	0	512	0	0	0
6	0	0	512	0	0	0	0	512	0	0	0
7	0	0	512	0	0	0	0	512	0	0	0
8	0	0	512	0	0	0	0	512	0	0	0
9	0	0	512	0	0	0	0	512	0	0	0
...											
15	0	0	512	3384	3266	6738	0	512	3422	3283	6555
16	0	**16384**	512	0	0	0	**16384**	512	0	0	0
17	0	**16384**	512	0	0	0	**16384**	512	0	0	0
18	0	**16384**	512	0	0	0	**16384**	512	0	0	0
19	0	**16384**	512	0	0	0	**16384**	512	0	0	0
20	0	**16384**	512	0	0	0	**16384**	512	0	0	0
21	0	**16384**	512	0	0	0	**16384**	512	0	0	0
22	0	**16384**	512	0	0	0	**16384**	512	0	0	0
23	0	**13324**	512	0	0	0	**13324**	512	0	0	0
24	0	0	512	0	0	0	0	512	0	0	1
...											

TABLE VIII
PART OF THE SPARE AREA FREQUENCY TABLE

TABLE IX
PHYSICAL BLOCK NUMBERS VERSUS SPARE AREA BLOCK NUMBER INFO

Physical Block	Block number in zone
0001	0001
041F	0001
0804	0001
0C25	0001
00F8	0002
0401	0002
0808	0002
0C26	0002
0004	0004
0423	0004
0805	0004
0C27	0004

spare area byte location.

Table VIII is a fragment of the total frequency table of the spare area's. In the columns 7 and 12 there is an even distribution of all possible values[28]. This is a good indication that there is a counter in this byte. The columns 6 and 11 only have values between 16 (0x10) and 23 (0x17)[29], so this might also be a counter, but only of 3 bits (bits 0 through 2) and bit 4 always high. Further examination of data in columns 6/11 and 7/12 learned that in all spare areas the data in these columns is the same. Apparently the counter is stored twice in spare areas. Data in column 6/11 and 7/12 adds up to an 11 bit counter, that can address 2^{11} erase blocks, which is 2048 * 16384 = 32 MByte. Next thing that has to be investigated is whether the counters are contiguous. If not, the erase blocks within a zone have to be arranged in the order of the counter in bytes 6 and 7, but missing numbers are allowed. Of course this will decrease the addressable memory space of the counter, and increase the number of zones. To find out about the contiguousness of the counter, make a table with physical block number and block number based on byte 6/7 of the spare area.

When ordered on the last column, it is clear that each block number appears 4 times within the whole memory, leading to the conclusion that there must be four zones. Table XI illustrates how to calculate logical block numbers from the

TABLE X
START AND ENDING BLOCKS OF ZONES

Zone	Starting Block	Ending block
0	0x000	0x3FF
1	0x400	0x7FF
2	0x800	0xBFF
3	0xC00	0xFFF

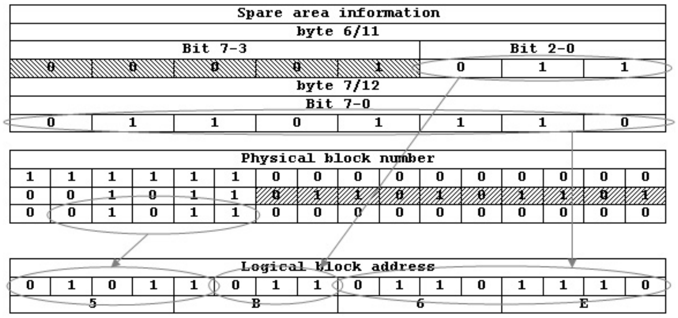


TABLE XI
HOW TO CALCULATE THE LBN

counter in byte 7/12 and bits 0-2 of byte 6/11 and the zone number. Expressed in C code this looks like:

```
Logicalblocknumber  =  Byte7;
Logicalblocknumber += (Byte6 & 0x07) << 8;
Logicalblocknumber += (Physicalblocknumber & 0xFC00) <<1;
```

In the example above the LBN is 0x5B6E. All blocks can now be rearranged by ordering the blocks by their LBN. When done, the result can be checked by calculating and comparing hash values over the reconstructed file system and over the image obtained through 'normal' methods.

B. Mobile Phone File System Analysis

Figure 16 depicts the NAND array structure of the multi-chip package memory in a Samsung SGH-D500 phone as described in the datasheet [17]. Four 512 byte data sectors are grouped into one page together with four 16 byte spare area data. Figure 17, also from [17] explains the assignment of the spare area bytes. The spare area description suggests that *logical sector numbers* (LSN) should be stored in byte 3-6 of each 16 byte spare area part. To verify this and determine the storage format a sample flash memory file with known data has been studied. The LSN is stored in byte 3-5 with the least significant byte first as demonstrated below:

```
FFFF 5C35 00FE 00FF 0C03 CCFF FFFF FFFF
LSN = 00355Ch = 13660d

FFFF 5D35 00FE 00FF 0C03 CCFF FFFF FFFF
LSN = 00355Dh = 13661d
```

Calculating all LSN's for an experimental flash memory file gives different physical sectors with the same LSN. Additional experiments were done to find out which physical sector from a set of sectors with identical LSN's is the sector that belongs

---

[28] Except for the values 0 and 255.

[29] Except for the values 0 and 255.

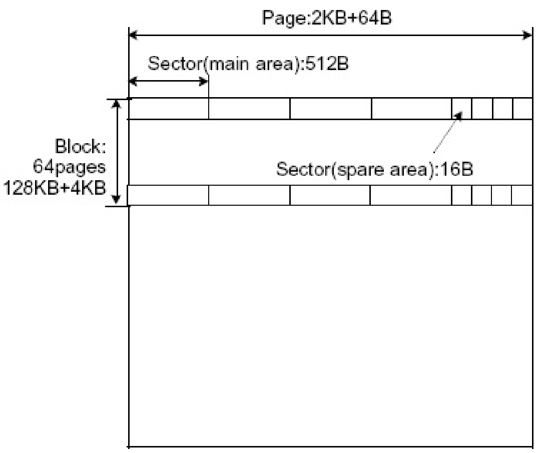
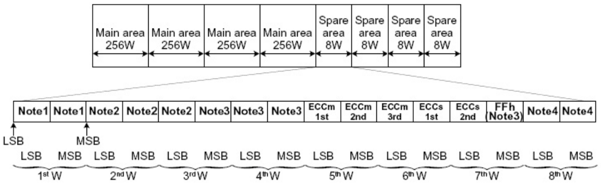


Fig. 16. NAND array structure of the multi-chip package memory in a Samsung SGH-D500 phone.



NOTE:
1) The 1st word of spare area in 1st and 2nd page of every invalid block is reserved for the invalid block information by manufacturer.
2) These words are managed by internal ECC logic. So it is recommended that the important data like LSN(Logical Sector Number) are written.
3) These words are reserved for the future purpose by manufacturer. These words will be dedicated to internal logic.
4) These words are for free usage.
5) The 5th, 6th and 7th words are dedicated to internal ECC logic. So these words are only readable. The other words are programmable by command.
6) ECCm 1st, ECCm 2nd, ECCm 3rd: ECC code for Main area data
7) ECCs 1st, ECCs 2nd: ECC code for 2nd and 3rd word of spare area.

Fig. 17. Assignment of spare area bytes

to the actual high level file system[30]. The sleuth kit *fsstat* and *fls* tools [19] were used to verify the different LSN hypotheses. Besides the LSN other info proved to be needed related to the 128KB+4KB block of which a sector belongs to (see figure 16). Each block starts with a special sector not belonging to the high level file system. This sector, starting with the tag "XSR" contains a 4 byte *block number* (BN) starting on the 21'th byte and a 4 byte *block version* (BV) starting on the 17'th byte, as demonstrated below:

```
5853 5231 6400 0000 0F00 0000 0100 0000 XSR1d...........
0701 0000 0100 0000 FC00 0000 0000 0000 ................
BN = 00000001h = 1d
BV = 00000107h = 263d
```

For physical sectors with identical LSN's the physical sector with the highest physical address must be used within the block with the highest BV. The findings have been used to implement a python script that builds a list *ListLSN* containing items for each encountered LSN. Each list item is also a list containing all physical addresses of sectors with a specific LSN, sorted on BV and physical address within a BV. A *ListLSN* example fragment:

```
ListLSN[23]={0x01264f20,0x01f77180}
ListLSN[24]={0x01265550,0x01265340,0x01265130,0x01f77390}
ListLSN[25]={0x01265760,0x01f775a0}
```

1) Reconstructing the high level file system: After building this list extracting the actual high level file system is trivial: for each LSN in *ListLSN* append 512 bytes (one data sector) from the flash memory file starting from the first physical address in *ListLSN[LSN]* (the light shaded addresses above) to a file storing the high level file system. If a LSN is not present in *ListLSN* add 512 dummy bytes instead[31].

2) Recovering other high level file system data: Not all flash memory data has been used to reconstruct the high level file system. The remaining data can be related to older instances of the high level file system[32] or used by the embedded systems firmware for other purposes[33]. For data carving purposes it is preferred to put all remaining data sectors in such an order to maximize the carving results. For sectors that belonged to a high level file system it might be a good strategy to put the sectors in the same logical order as before they were deleted.

The *ListLSN* list introduced before contains information that can be used for this purpose. If instead of all first addresses of each *ListLSN* item the next addresses are used (the dark shaded addresses in the *ListLSN* fragment) another high level file system can be generated with older data. Because it is not known when sectors are deleted and not all deleted sectors will be available[34], the reconstructed high level file system will most likely not be fully consistent but for data analysis purposes it will work.

In the python script mentioned before the following heuristic approach has been used to extract all data that is not part of the actual high level file system:

1) For each LSN item in ListLSN remove the first address[35]
2) For each LSN item with less address items than a certain threshold:
    a) Initialize the currents address item to the first address of the first LSN item.
    b) Export the current address item of the current LSN item and remove it from *ListLSN*. Then look at all address items of the next LSN item and try to find the best match. The following heuristic is used for the matching:
        i) If the absolute difference between the current LSN item's address and the next LSN item's

---

[30] After searching the NAND flash memory file for file system specific tags it looks like FAT16 is used as high level file system.

[31] Not every LSN is necessarily present in the flash memory file.

[32] For example to logical sectors belonging to a file that has been deleted and which LSN has been reused by the high level file system. Especially for small data objects with a high refresh frequency (e.g. FAT, directory entries), a lot of old versions might exist in the flash memory file.

[33] For example as workspace for the flash file system during flash memory clean-up operations.

[34] Sectors that are not part of the current high level file system can be overwritten by the flash file system at any time.

[35] This address has been used to generate the actual high level file system.

```
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: FAT16

OEM Name: MSWIN4.1
Volume ID: 0xcccc
Volume Label (Boot Sector): KFAT0Volume Label (Root Directory):
KFAT0File System Type Label: FAT16

Sectors before file system: 0

File System Layout (in sectors)
Total Range: 0 - 248387
* Reserved: 0 - 0
** Boot Sector: 0
* FAT 0: 1 - 122
* FAT 1: 123 - 244
* Data Area: 245 - 248387
** Root Directory: 245 - 276
** Cluster Area: 277 - 248380
** Non-clustered: 248381 - 248387

METADATA INFORMATION
--------------------------------------------
Range: 2 - 3969666
Root Directory: 2

CONTENT INFORMATION
--------------------------------------------
Sector Size: 512
Cluster Size: 4096
Total Cluster Range: 2 - 31014

FAT CONTENTS (in sectors)
--------------------------------------------
277-284 (8) -> EOF

...

68597-68604 (8) -> EOF
```

Fig. 18. Output of TSK command: “fsstat -o4 -f fat D500_FATFS.bin”

address is exactly the sector size this address is selected and no other addresses are matched.

ii) If the current LSN item address is the last data sector of a block and the next LSN item's address is the first data sector of a block this address is selected and no further matching is done.

iii) If no match has been made, don't pick an address but restart at *a*. with the first address of the first item in *ListLSN*.

c) If a match is found go to *b*. with the matched address as current item's address of the current LSN item.

3) Step 2. is repeated until no LSN items exist with less address items than the threshold value.

4) All remaining sectors from *ListLSN* are selected and there related data sectors are exported.

The described method does not guarantee the most optimal sector order for data analysis applications, it's just a quick heuristic that can possibly be improved when more knowledge about the flash file system dynamics are known.

3) Mobile phone FAT file system analysis: The file with the reconstructed high level file system (see section IV-B1) can be further analyzed with any forensic disk analysis tool. Figure 18 gives the output of The Sleuth Kit (TSK) [19] *fsstat* tool. Figure 19 shows the root directory of the FAT-FS displayed with TSK command *fls*.

```
d/d 5:  sounds
d/d 6:  images
d/d 7:  user
d/d 8:  sms
d/d 9:  test
d/d 10: mms
d/d 11: java
d/d 13: multimedia
d/d 14: drm
r/r 17: tfsVersionCode.tfs
```

Fig. 19. Output of TSK command “fls -o4 -ffat D500_FATFS.bin”

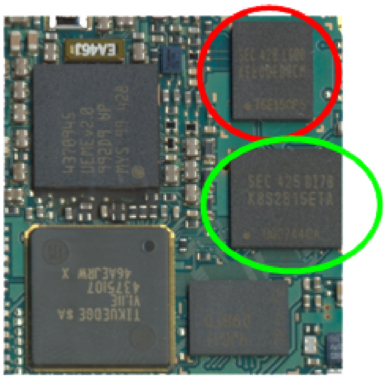


Fig. 20. The flash chips on the main board

Further information on common file system forensic analysis can be found in the reference section. In chapter 5 some flash specific issues of file system forensic analysis will be discussed.

4) Results on a Nokia 6230: In this paragraph the reference model phone is a Nokia 6230 with the Series 40 2nd edition operating system. The two flash chips that were found in this phone are: Samsung K8S2815ETA 64Mbit NOR flash and Samsung KEE00E00CM 64Mbit NAND flash. In Figure 20 the NAND flash is encircled red and the NOR flash is encircled green. Both chips have been physically removed and read with the method described in section III-C.

Multimedia data is stored in the NAND flash which is organized in 996 blocks of 8448 bytes[36]. Each block contains 512 bytes sized data sectors with a spare area of 16 bytes. The first sector of each block starts with the string “SSR200” and does not contain user data. In this 512 byte block header the bytes {25...24}give the block number. After each sector of 512 bytes are 10 spare area bytes. The bytes {0...1} of this spare are contain the logical sector number. Unused sectors have 0xffff as sector number.

Each block can contain multiple sectors with the same LSN. The sector with the highest physical address in a block is the valid sector. To reconstruct the FAT FS from the memory copy, invalid sectors need to be removed; all other sectors need to be put in the right order and for missing LSN's numbers 512 bytes of dummy data need to be inserted. All data not belonging to the (current) file system can be used for data carving purposes.

5) Results on Symbian phones: According to [27], the drive, directory and file hierarchy on the flash memory of phones running Symbian complies with VFAT[37] specifications, making the file system compatible with desktop PC's. The actual data itself however is stored in a proprietary format. The results described in this paragraph are based on experiments with Nokia 3650 and 7610 phones [26]. The described block and page headers correspond to a Nokia 7610 phone, for the 3650 they are some differences in sizes and offsets.

---

[36] The remainder of the 8 MByte can be found at the beginning and at the end of the chip's address space but has no recognizable block structure.

a) Data acquisition: The flash memory of the Nokia 3650 can be copied by using the *RRawDisk* function of the Symbian file server API. Because this function only works when no other resources are accessing the file system, the Symbian backup server was used to close all handles to open files before using the *RRawDisk* function. The application to copy the internal flash memory needs to be installed on an external MMC card which is also used as destination for the flash memory copy. Unfortunately this method did not work on other Symbian phones because not all file handles could be closed successfully. The flash memories of the Nokia 7610 have been removed physically and copied with the NFI memory toolkit as described in section III-C.

b) Block and sector headers: The memory is divided in 128 blocks of 64 kb each. A block starts with a 76 bytes block header, which holds information about the block. The rest of the block contains data sectors and sector headers. Each data sector has a 27 bytes wide header located just after the block header header. Data sectors are written to a block in a "bottom-up" fashion, and can have a maximum size of 512 bytes. Sequences of one or more data sectors constitute a stream. A single stream can represent any kind of data, for example, a file or a directory table. The data sectors that a stream consists of need not be stored in a sequential manner, and can span multiple blocks. The size and position of a data sector is determined by its sector header. The sector header also determines which stream the data block belongs to, and the sequence number of the data-block.

If a block is empty, only the block header exists. Otherwise, the block header is first followed by a sector header that does not point to any data sector. Then a list of 'useful' data sector headers follows. The list of sector headers is of constant size. Table XII and table XIII contain partial specifications (determined from experiments) of the fields of block and sector headers.

When a file is deleted the sectors of a stream become dirty, this is achieved by setting the valid bits in the sector header to 0. The sector still exists in the flash device, but is marked as dirty.

c) Files and directories: A file consists of two streams, with the same stream ID. Only the data type field in the sector header (table XII) will be different. 0x84 in this field denotes that the stream represents the contents of the file itself, while 0x81 means that the stream contains the file attributes of this file. The only known field in this stream is the 6th byte, which denotes whether the data stream represents a user file, or directory: 0x01 means the stream represents a file, 0x02 means the stream represents a directory. 0x03 indicates that this stream is the root-directory. A directory is stored in much the same way as a normal file. The content of this file is a directory table, with a list directory entries pointing to its files and subdirectories. A directory entry is organized as depicted in table XIV.

d) Decoding a memory copy: With the information described in the previous section it is possible to decode the current file system. Deleted files can be recovered by looking for stream id's in dirty sectors but this will be complicated if different deleted files with the same stream id exist.

Byte	Meaning
{3...0}	The block number, (0-63)
{7...4}	After formatting, all blocks are numbered 1 to 64. The first dirty page that is to be erased after a format will be labeled 65, the second 66, etc This number is perhaps used by a wear-leveling algorithm
{11...8}	Initially contains the same number as 3..0, but is incremented after the block becomes dirty and after being overwritten. Possibly also related to wear-leveling
{15...12}	Initially contains the same number as 3..0, but is incremented after the block becomes dirty and after being overwritten. Possibly also related to a wear-leveling
{12...43}	Seems to be same for every block
{75...72}	Possibly a checksum of the block header

TABLE XII
INTERPRETATION OF BYTES 0-75 OF A BLOCK HEADER

Byte	Meaning
{0}	Data sector attributes; bit 2 and bit 3 of this byte (bit 0=LSB) seems to indicate whether the data sector contains valid data, or whether it is dirty
{1}	The type of data that this stream represents
{2}{3}	Unknown
{5...4}	Stream ID
{7...6}	Were always found to be zero The stream ID may be a 32-bit number, with {7..6} as the higher bytes, instead of a 16-bit number
{9...8}	Sequence number of the sector in the stream
{11...10}	Seems to be always zero, though the sequence number may be a 32-bit number, with these as the higher bytes
{13}{12}	This is an address offset of the data sector. The offset is relative to the start of the block. This address is a word-address, so a left-shift is needed to obtain the proper byte-address
{15}{14}	Size of the data sector in bytes
{19...16}	Possibly a checksum of the data sector
{23...20}	Possibly a checksum of the page sector

TABLE XIII
INTERPRETATION OF BYTES 0-23 OF A SECTOR HEADER

Byte	Meaning
{3...0}	Length of current directory entry in bytes. This is always a multiple of 4 bytes
{7...4}	Stream id of the relevant file or subdirectory
{11...8}	Some sort of counter; zero for the first directory entry; the next entry is always 3-5 higher than the previous. The meaning of this field is unknown
{12...size-of-entry}	Name of the file/subdirectory, in 16-bit unicode. Because the length of a directory entry is always a multiple of four, the name will be padded with 0x0000 if the amount of characters in the name is an odd number

TABLE XIV
THE INTERPRETATION OF A DIRECTORY ENTRY

---

[37] Virtual File Allocation Table, a virtual installable files system driver serving as an interface between applications and the File Allocation Table (FAT).

V. EVIDENCE SEARCHING

After reconstruction of the high level file system and extraction of all other data in the most plausible order, further investigation of the flash memory can be done in the same way as for any other forensic file system investigation [3]. This section will discuss some specific analysis topics related to data originating from flash file systems.

A. File System Tools

Current forensic file system analysis tools like *Encase, FTK, R-Studio, and TSK*, are not fully aware of the physical media from which an image file originates. For advanced data recovery this knowledge of the physical properties might improve the recovery process. Flash file systems for example often contain different versions of the same data objects because flash memory can't be erased in small quantities. Especially for small objects (much smaller than one flash block) with a high update frequency, a lot of old versions might exist outside of the normal high level file system.

For FAT file systems the FAT and directory entries are interesting candidates for advanced analysis because of their size, update frequency and evidence value. To give an idea of the amount of different versions: in an actual case with a Samsung SGH-D500 mobile phone the flash memory file contained 83 versions of some part of the FAT and 1464 versions of the directory "\multimedia\VIDEOS\video clips" were all user recorded video movies are stored by default. A common forensic tool will show the last version of the directory, possibly with some files marked as deleted but from the other versions of the directory data a lot of the user behavior can be reconstructed.

The same holds for other data objects although larger objects (like movie files) are likely to be (partly) overwritten earlier after deletion because they occupy complete flash blocks which can be reused immediately after deletion.

B. Dedicated Search Strategies

Some specific flash memory analysis issues are discussed below based on a case example. In this case a doubtful witness declares that he made a recording with his phone on which somebody confessed a murder. A standard forensic investigation of the mobile phone with .XRY [21] did not show any relevant data so it was sent to the NFI for advanced analysis on the presence of erased audio or video material.

At the laboratory the flash memory chip was analyzed of a reference phone of the same brand and type. It contains 32 MByte of NOR flash and 128 MB of NAND flash. From experiments it was found that multimedia data like sound, pictures and video are stored in the NAND flash memory so this memory was further examined on the case phone. JTAG [10] has been used to copy the NAND flash data to a binary file. This file has been used with the script discussed in section IV-B1 to extract a file with the FAT file system and a file with the remaining data in the most plausible order as discussed before.

R-Studio has been used to load the FAT file system file and recover all file system data to a file server. With R-Studio three erased video files could be identified in the "video clips" folder. The recover option of R-Studio reported that differences between the file size and the length of the FAT chain indicate that these files were overwritten. The video clips folder contains a "THUMB" folder with the same file entries but shorter file sizes. R-Studio reported that these files were successfully recovered but because it was not possible to decode these files they were further analyzed with TSK. With the *fls* tool the cluster number of each file was decoded from the directory entries and with the *istat* tool all other file metadata was displayed. The *istat* tool reported that file recovery was not possible. How these files can still be recovered by using data sectors from the flash memory not present in the FAT file system is illustrated with example thumbnail "video-0003.3gp". fls -o4 -f fat -r D500_FATFS.bin gives for "video-0003.3gp":

```
++++ r/r * 5901:        video-0003.3gp
```

istat -o4 -f fat D500_FATFS.bin 5901 gives:

```
Directory Entry: 5901
Not Allocated
File Attributes: File
Size: 2720
Name: _IDEO-~3.3gp
Directory Entry Times:
Written:         Tue May  3 17:41:24 2005
Accessed:        Tue May  3 00:00:00 2005
Created:         Tue May  3 17:41:24 2005
Sectors:
88909 88910 88911 88912 88913 88914 88915 88916
Recovery:
File recovery not possible
```

Looking in *ListLSN* for the sector numbers gives[38]:

```
ListLSN[88913]={0x06a7c940,0x044515b0,0x0444e430,0x067e0540}
ListLSN[88914]={0x06a7cb50,0x044517c0,0x0444e640,0x067e0330}
ListLSN[88915]={0x06a7cd60,0x044519d0,0x0444e850,0x067e0120}
ListLSN[88916]={0x06a7cf70,0x04451be0,0x0444ea60,0x067dff10}
ListLSN[88917]={0x06a7d180,0x04451df0,0x0444ec70,0x067dfd00}
ListLSN[88918]={0x06a7d390,0x0444ee80,0x067dfaf0}
ListLSN[88919]={0x06a7d5a0,0x0444f090}
ListLSN[88920]={0x06a7d7b0,0x0444f2a0}
```

The current FAT-FS uses address range 0x06a7c940-(0x06a7d7b0+0x200) for logical sectors 88913-88920 but that data range is currently allocated to another file. If the non FAT-FS sectors are grouped according to the heuristic described in section IV-B2 the following ranges appear in the resulting file:

```
Range 1: 0x044515b0,0x044517c0,0x044519d0,0x04451be0,
         0x04451df0
Range 2: 0x0444e430,0x0444e640,0x0444e850,0x0444ea60,
         0x0444ec70,0x0444ee80,0x0444f090,0x0444f2a0
Range 3: 0x067e0540,0x067e0330,0x067e0120,0x067dff10,
         0x067dfd00,0x067dfaf0
```

After comparing data from known thumbnail video files the data belonging to range 3 could be decoded to a valid picture file showing a small thumbnail of a video clip as used by the multimedia browser on the phone. From the decoded thumbnails two of the related video files could be marked as “not relevant”.

---

[38] 4 sectors need to be added to *istat's* starting sector because the flash manager takes the first offset sectors into account and TSK not.

In the search for deleted audio and video data a script was developed to use the non erased data to find all data in the flash memory file originating from this non erased data . The script produces a bookmark file for the Hex Workshop hex editor [22] to make it easy to find all the known data parts. With another script this bookmark file can be used to overwrite all known data with a predefined pattern in order to make searching for erased data easier. To assist the manual search for deleted audio and video data a tool has been used developed by the NFI to analyze video data. For this tool a .3gp parser has been written to search for data fragments originating from .3gp files. After extensive search no audio or video data has been found that confirmed the statement of the witness.

VI. CONCLUSION

Three techniques have been described for making low level byte-by-byte copies of flash memory chips. More research needs to be done on the flash read mechanisms used by flasher tools in order to adapt these mechanisms for usage in the next generation of forensic data acquisitions tools. Steps have been illustrated for translating acquired flash data to a level that can be understood by existing forensic tools targeted towards common used file systems. More research is needed for flash data that can’t directly be translated to file system level. More research is also needed on the relation between flash specific operations like block erasing and wear leveling on one side and the resulting artifacts and potentials for data recovery and analysis on the other side. With the results of this research future forensic tools might be able to improve the power and efficiency of embedded systems examinations for reasonably skilled IT professionals.

VII. ABBREVIATIONS

ATA - Advanced Technology Attachment
API - Application Programming Interface
BGA - Ball Grid Array
CFI - Common Flash Interface
ECC - Used for both Error Correcting Code and Error Checking and Correction.
EEPROM - Electrically Erasable Programmable Read Only Memory
FAT - File Allocation Table
FFS - Flash File System
FSD - File System Driver
FTL - Flash Translation Layer
I/O - Input/Output
JTAG - Joint Test Action Group
LBA - Logical Block Address
LBN - Logical Block Number
LSN - Logical Sector Number
MMC- Multi Media Card
NFI - Netherlands Forensic Institute
OS - Operating System
PCB - Printed Circuit Board
PDA - Personal Data Assistent
RAPI - Remote Application Programming Interface
SCSI - Small Computer System Interface
TSK - The Sleuth Kit
TSOP - Thin Small-Outline Package
USB - Universal Serial Bus
VFAT - Virtual File Allocation Table
XSR - Extended Sector Remapper
ZIF - Zero Insertion Force

REFERENCES

[1] R. van der Knijff, “Embedded Systems Analysis”, chapter 11 of “Handbook of Computer Crime Investigations - Forensic Tools and Technology” edited by E. Casey. Academic press, 2002.

[2] W. Jansen and R. Ayers, “Guidelines on Cell Phone Forensics,” August 2006. [Online]. Available: http://csrc.nist.gov/publications/drafts/Draft-SP800-101.pdf. [Accessed: November 29, 2006]

[3] B. Carrier, *File System Forensic Analysis*. Addison-Wesley 2005.

[4] E. Sutter, *Firmware Demystified*. CMP Books, 2002.

[5] Samsung Electronics, “APPLICATION NOTE for NAND Flash Memory”, rev, 2, 1999. [Online]. Available: http://www.samsung.com/Products/Semiconductor/Memory/appnote/app_nand.pdf. [Accessed: November 29, 2006].

[6] Sandisk, “Sandisk flash memory cards - wear leveling”, October 2003. [Online]. Available: www.sandisk.com/Assets/File/OEM/WhitePapersAndBrochures/RS-MMC/WPaperWearLevelv1.0.pdf. [Accessed: November 29, 2006].

[7] M-Systems, “TrueFFS wear-leveling Mechanism”, *Technical note (TN-Doc-017)*. [Online]. Available: www.m-systems.com/NR/rdonlyres/FCC7D817-38A5-4D80-8471-67DA793EA255/0/TN_017_TrueFFS_Wear_Leveling_Mechanism.pdf. [Accessed: November 29, 2006].

[8] HDDGURU, “ATA/ATAPI Command Set”, [Online]. Available: http://hddguru.com/content/en/documentation/2006.01.27-ATA-ATAPI-8-rev2b/. [Accessed: November 29, 2006].

[9] Samsung Electronics, “Smartmedia Format Introduction (Software Considerations)”, 1999. [Online]. Available: www.win.tue.nl/~aeb/linux/smartmedia/SmartMedia_Format.pdf.

[10] B. Dipert and M. Levy, “Designing with Flash Memory,” Annabooks, 1994.

[11] M. Breeuwsma, “Forensic imaging of embedded systems using JTAG (boundary-scan)”, *Digital Investigation*, vol. 3, ed. 1, March 2006.

[12] IEEE - Standards Association, “IEEE standard test access port and boundary scan architecture - description”, July 23, 2001. [Online]. Available: http://standards.ieee.org/reading/ieee/std_public/description/testtech/1149.1-2001_desc.html.

[13] Intel, “Intel Wireless Communications and Computing Package User’s Guide”, [Online]. Available: http://download.intel.com/design/flcomp/packdata/wccp/25341805.pdf. [Accessed: November 29, 2006].

[14] PACE, [Online]. Available: http://www.paceworldwide.com/. [Accessed: November 30, 2006].

[15] Signum, [Online]. Available: http://www.signum.com. [Accessed: November 30, 2006].

[16] BPM Microsystems, [Online]. Available: http://www.bpmicro.com/. [Accessed: November 30, 2006].

[17] Samsung Electronics, “Datasheet of the Multi-Chip Package MEMORY, 256M Bit (16M x16) Synchronous Burst , Multi Bank NOR Flash Memory / 512M Bit(32Mx16) OneNAND, Flash*2 / 128M Bit(8Mx16) Synchronous Burst Uni-Transistor Random Access Memory”, [Online]. Available: http://samsung.com/.

[18] Logic Technology, “Universal socket solution”, [Online]. Available: http://www.logic.nl/trial.aspx?sid=380(ZIPpassword=”FFIT-2006”). [Accessed: November 30, 2006].

[19] The Sleuth Kit, [Online]. Available: http://www.sleuthkit.org. [Accessed: November 29, 2006].

[20] rtt, “R-Studio Data Recovery Software”, [Online]. Available: http://www.data-recovery-software.net/. [Accessed: November 29, 2006].

[21] Micro Systemation, “.XRY phone examination”, [Online]. Available: http://www.msab.com/en/. [Accessed: November 29, 2006].

[22] BreakPoint Software, “Hex Workshop, a set of hexadecimal development tools for Microsoft Windows”, [Online]. Available: http://www.hexworkshop.com/. [Accessed: November 29, 2006].
[23] GSM-Forum, [Online]. Available: http://forum.gsmhosting.com/vbb/index.php. [Accessed: November 29, 2006].
[24] Multi-com, [Online]. Available: http://www.gsm-technology.com. [Accessed: November 29, 2006].
[25] GSM-Technology, [Online]. Available: http://www.gsm-technology.com. [Accessed: November 29, 2006].
[26] K. Li, “Datarecovery on a Nokia 3650”, *NFI internal report*, August 2005.
[27] Symbian, “Symbian Developer library”, [Online]. Available: http://www.symbian.com/developer/techlib/sdl.html. [Accessed: November 29, 2006].

Marcel Breeuwsma Marcel Breeuwsma received a BSc in computer science from the ‘The Hague University’. In 1996 he joined the Netherlands Forensic Institute to work in the field of R&D. His main interests are digital electronics, embedded processors, embedded software and FPGA chips. He developed a Memory Tool Kit including several versions of software, builds hardware for an organizer analysis tool and did research in using JTAG for forensic purposes.

Martien de Jongh Martien de Jongh has an educational background in electrical engineering and electronics and a lot of experience in consumer electronics. He started to work at the Netherlands Forensic Institute in 1998 where he first worked on acquisition and decoding of data from mobile phones and is currently specialized in physical data acquisition from digital electronics.

Coert Klaver Coert Klaver received his B.Sc. in computer science in 1986 from the s-Hertogenbosch College of Advanced Technology. He has worked for several public organizations and private companies since then, developing embedded software and hardware. In 1999 he joined the Netherlands Forensic Institute to work as a forensic scientist in the field of embedded systems. His main interests are embedded operating systems in personal digital electronics and improvised electronics used in high tech fraud schemes.

Ronald van der Knijff Ronald van der Knijff received his B.Sc. degree on electrical engineering in 1991 from the Rijswijk Institute of Technology. After performing military service as a Signal Officer he obtained his M.Sc. degree on Information Technology in 1996 from the Eindhoven University of Technology. Since then he works at the Digital Technology and Biometrics department of the Netherlands Forensic Institute as a scientific investigator. He is responsible for the embedded systems group and is also court-appointed expert witness in this area. He is author of the (outdated) cards4labs and TULP software and founder of the TULP2G framework. He is a visiting lecturer on ‘Cards & IT’ at the Dutch Police Academy; a visiting lecturer on ‘Smart Cards and Biometrics’ at the Masters Program ‘Information Technology’ of TiasNimbas Business School and a visiting lecturer on ‘Mobile and Embedded Device Forensics’ at the Master’s in ‘Artificial Intelligence’ of the University in Amsterdam (UvA). He wrote a chapter on embedded systems analysis in Eoghan Casey’s Handbook of Computer Crime Investigation - Forensic Tools and Technology.

Mark Roeloffs Mark Roeloffs received his B.Eng. degree on electrical engineering in 2004 from the Rotterdam University (a university of Applied Sciences). Since then he has worked at the Digital Technology and Biometrics department of the Netherlands Forensic Institute (NFI) as a forensic examiner. At NFI, Mark investigate mobile telephones for deleted data. He has also presented the course ‘Reading GSM Telephones for Investigators’ at the Dutch Police Academy.

View publication stats